

Parametric Polymorphism and Semantic Subtyping: the Logical Connection

Nils Gesbert

INRIA
 nils.gesbert@inria.fr

Pierre Genevès

CNRS
 pierre.geneves@inria.fr

Nabil Layaïda

INRIA
 nabil.layaïda@inria.fr

Abstract

We consider a type algebra equipped with recursive, product, function, intersection, union, and complement types together with type variables and implicit universal quantification over them. We consider the subtyping relation recently defined by Castagna and Xu over such type expressions and show how this relation can be decided in EXPTIME, answering an open question. The novelty, originality and strength of our solution reside in introducing a logical modeling for the semantic subtyping framework. We model semantic subtyping in a tree logic and use a satisfiability-testing algorithm in order to decide subtyping. We report on practical experiments made with a full implementation of the system. This provides a powerful polymorphic type system aiming at maintaining full static type-safety of functional programs that manipulate trees, even with higher-order functions, which is particularly useful in the context of XML.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Polymorphism; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Algorithms, Design, Languages, Theory, Verification

Keywords Type-system, Polymorphism, subtyping

1. Introduction

This paper studies parametric polymorphism for type systems aiming at maintaining full static type-safety of functional programs that manipulate linked structures such as trees, potentially with higher-order functions. We consider a type algebra equipped with recursive, product, function (arrow), intersection, union, and complement types. We first show how the subtyping relation between such type expressions can be decided through a logical approach. Our main result solves an open problem: we prove the decidability of the subtyping relation when this type algebra is extended with type variables. This provides a powerful polymorphic type system (using ML-style prenex polymorphism, where variables are implicitly universally quantified at toplevel), for which defining the subtyping relation is not obvious, as pointed out in [5] and discussed in

Section 5.1, and for which no candidate definition of subtyping had been proved decidable before. The novelty, originality and strength of our solution reside in introducing a logical modeling for the semantic subtyping framework. Specifically, we model semantic subtyping in a mu-calculus over finite trees and rely on a satisfiability solver in order to decide subtyping in practice. We obtain an EXPTIME complexity bound as well as an efficient implementation in practice.

1.1 The Need for Polymorphism and Subtyping

Subtyping makes it possible to prove that term substitution in a program source code preserves type-safety. For example, let us consider a simple property relating polymorphic types of functions that manipulate lists. We consider a type α , and denote by $[\alpha]$ the type of α -lists (lists whose elements are of type α). The type τ of functions that process an α -list and return a boolean is written as follows:

$$\tau = [\alpha] \rightarrow \text{Bool}$$

where $\text{Bool} = \{\text{true}, \text{false}\}$ is the type containing only the two values `true` and `false`. Now let us consider functions that distinguish α -lists of even length from α -lists of odd length: such a function returns `true` for lists with an even number of elements of type α , and returns `false` for lists with an odd number of elements of type α . One may represent the set of these functions by a type τ' written as follows:

$$\begin{aligned} \text{even}[\alpha] &\rightarrow \{\text{true}\} \\ \wedge \text{odd}[\alpha] &\rightarrow \{\text{false}\} \end{aligned}$$

where $\{\text{true}\}$ and $\{\text{false}\}$ are singleton types (containing just one value). If we make explicit the parametric types $\text{even}[\alpha]$ and $\text{odd}[\alpha]$, τ' becomes:

$$\begin{aligned} \tau' = & \quad \mu v. (\alpha \times (\alpha \times v)) \vee \text{nil} && \rightarrow \{\text{true}\} \\ \wedge & \quad \mu v. (\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil}) && \rightarrow \{\text{false}\} \end{aligned}$$

where \times denotes the cartesian product, μ binds the variable v for denoting a recursive type, and `nil` is a singleton type.

Obviously, a particular function of type τ' can also be seen as a less-specific function of type τ . In other terms, from a practical point of view, a function of type τ can be replaced by a more specific function of type τ' while preserving type-safety (however the converse is not true). This is further formalized by the notion of subtyping; in that case we write:

$$\tau' \leq \tau \tag{1}$$

where \leq denotes a subtyping relation that can be defined in two fundamentally different ways in the literature: either syntactically or semantically. In this paper, we define \leq as a semantic subtyping relation by adopting a set-theoretic interpretation in the manner of [9], in contrast with more traditional subtyping through direct



syntactic rules. As a main contribution, we show how to decide this relation.

This work is motivated by a growing need for polymorphic type systems for programming languages that manipulate XML data. For instance, XQuery [4] is the standard query and functional language designed for querying collections of XML data. The support of higher-order functions, currently missing from XQuery, appears in the requirements for the forthcoming XQuery 3.0 language [8]. This results in an increasing demand in algorithms for proving or disproving statements such as the one of the example (1) with polymorphic types, but also with types of higher-order functions (like the traditional `map` and `fold` functions), or more generally, statements involving the subtyping relation over a type algebra with recursive, product, function, intersection, union, and complement types together with type variables and universal quantification over them.

1.2 Semantic Subtyping with Logical Solvers

During the last few years, a growing interest has been seen in the use of logical solvers such as satisfiability solver and satisfiability-modulo solvers in the context of functional programming and static type checking [1, 3]. In particular, solvers for tree logics [7, 10] are used as basic building blocks for type systems for XQuery.

The main idea in this paper is a type-checking algorithm for polymorphic types based on deciding subtyping through a logical solver. To decide whether τ is a subtype of type τ' , we first construct equivalent logical formulas φ_τ and $\varphi_{\tau'}$ and then check the validity of the formula $\psi = \varphi_\tau \Rightarrow \varphi_{\tau'}$ by testing the unsatisfiability of $\neg\psi$ using the satisfiability-testing solver. This technique corresponds to semantic subtyping [9] since the underlying logic is inherently tied to a set-theoretic interpretation. Semantic subtyping has been applied to a wide variety of types including refinement types [3] and types for XML such as regular tree types [12], function types [2], and XPath [6] expressions [10].

This fruitful connection between logics, their decision procedures, and programming languages permitted to equip the latter with rich type systems for sophisticated programming constructs such as expressive pattern-matching and querying techniques. The potential benefits of this interconnection crucially depend on the expressivity of the underlying logics. Therefore, there is an increasing demand for more and more expressiveness. For example, in the context of XML:

- SMT solvers like [7] offer an expressive power that corresponds to a fragment of first-order logic in order to solve the intersection problem between two queries [1];
- full first-order logic solvers over finite trees [10] solve containment and equivalence of XPath expressions;
- monadic second-order logic solvers over trees, and – equivalent yet much more effective – satisfiability-solvers for μ -calculus over trees [10] are used to solve query containment problems in the presence of type constraints.

1.3 Contributions of the Paper

To the best of our knowledge, novelty of our work is threefold. It is the first work that:

- proves the decidability of semantic subtyping for polymorphic types with function, product, intersection, union, and complement types, as defined by Castagna and Xu [5], and gives a precise complexity upper-bound: $2^{(n)}$, where n is the size of types being checked. Decidability was only conjectured by Castagna and Xu before our result, although they have now proved it independently; our result on complexity is still the only one. In

addition, we provide an effective implementation of the decision procedure.

- produces counterexamples whenever subtyping does not hold. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold.
- pushes the integration between programming languages and logical solvers to a very high level. The logic in use is not only capable to range over higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logic such as XML tree types [10]. This shows that such solvers can become the core of XML-centric functional languages type-checkers such as those used in CDuce [2] or XDuce [11].

1.4 Structure of the Paper

We introduce the semantic subtyping framework in Section 2 where we start with the monomorphic type algebra (without type variables). We present the tree logic in which we model semantic subtyping in Section 3. We detail the logical encoding of types in Section 4. Then, in Section 5 we extend the type algebra with type variables, and state the main result of the paper: we show how to decide the subtyping relation for the polymorphic case in exponential time. We report on practical experiments using the implementation in Section 6. Finally, we discuss related work in Section 7 before concluding in Section 8.

2. Semantic Subtyping Framework

In this section, we present the type algebra we consider: we introduce its syntax and define its semantics in terms of semantic domains. This framework is the one described at length in [9]; we do not discuss its properties here but just give the necessary definitions, that we will then extend with type variables in Section 5.

2.1 Types

Type terms are defined using the following grammar:

$\tau ::=$	b	basic type
	$\tau \times \tau$	product type
	$\tau \rightarrow \tau$	function type
	$\tau \vee \tau$	union type
	$\neg\tau$	complement type
	$\mathbf{0}$	empty type
	v	recursion variable
	$\mu v.\tau$	recursive type

We consider μ as a binder and define the notions of free and bound variables and closed terms as standard. A type is a closed type term which is *well-formed* in the sense that:

- the negation operator only occurs in front of *closed* types;
- every occurrence of a recursion variable is separated from its binder by at least one occurrence of the product or arrow constructor.

So, for example, $\mu v.\mathbf{0} \vee v$ is not well-formed, nor is $\mu v.\mathbf{0} \rightarrow \neg v$. Additionally, the following abbreviations are defined:

$$\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$$

and

$$\mathbf{1} = \neg\mathbf{0}$$

2.2 Semantic domain

Consider an arbitrary set \mathcal{C} of constants. From it, we define the semantic domain \mathcal{D} as the set of *ds* generated by the following

grammar, where c ranges over constants in \mathcal{C} :

$d ::=$		domain element
	c	base constant
	(d, d)	pair
	$\{(d, d'), \dots, (d, d')\}$	function
$d' ::=$	d	extended domain element
	Ω	error

The function terms are *finite* sets of pairs representing non-deterministic partial functions from \mathcal{D} to $\mathcal{D} \cup \{\Omega\}$: each pair (d, d') in the set means that, when given d as an argument, the function may yield d' as a result. If d does not appear as the first element of any pair, the operational interpretation is that the function can still accept d as an argument but will not yield a result: this represents a computation which does not terminate. A pair of the form (d, Ω) is used to represent a function rejecting d as an argument: when given d , it yields an error.

This grammar is only able to represent functions which diverge but on a finite number of possible arguments. However it is shown in [9] (Lemma 6.32) that considering only those functions does not affect the subtyping relation.

2.3 Interpretation

We suppose we have an interpretation $\mathbb{B}[\cdot]$ of basic types b as subsets of \mathcal{C} .

The predicate $(d' : \tau)$ where d' is an element of \mathcal{D} or Ω and τ is a type is defined recursively in the following way:

$$\begin{aligned}
(\Omega : \tau) &= \text{false} \\
(c : b) &= c \in \mathbb{B}[b] \\
((d_1, d_2) : \tau_1 \times \tau_2) &= (d_1 : \tau_1) \wedge (d_2 : \tau_2) \\
(\{(d_1, d'_1), \dots, (d_n, d'_n)\} : \tau_1 \rightarrow \tau_2) &= \forall i, (d_i : \tau_1) \Rightarrow (d'_i : \tau_2) \\
(d : \tau_1 \vee \tau_2) &= (d : \tau_1) \vee (d : \tau_2) \\
(d : \neg \tau) &= \neg(d : \tau) \\
(d : \mu v. \tau) &= (d : \tau\{\mu v. \tau/v\}) \\
(d : \tau) &= \text{false in any other case}
\end{aligned}$$

To prove this definition is well-founded, we first define the shallow depth of a type term as the longest path, in its syntactic tree, starting from the root and consisting only of μ , \vee , and \neg nodes. We then use the following ordering on pairs (d', t) :

- $d'_1 \leq d'_2$ if d'_1 is a subterm of d'_2
- $\tau_1 \leq \tau_2$ if the shallow depth of τ_1 is less than the shallow depth of τ_2
- pairs are ordered lexicographically, i. e. $(d'_1, \tau_1) \leq (d'_2, \tau_2)$ if either $d'_1 < d'_2$ or $d'_1 = d'_2$ and $\tau_1 \leq \tau_2$.

Now we can see that all occurrences of the predicate on the right-hand side of the definition are for pairs strictly smaller than the one on the left (in the case of $\mu v. \tau$, this is due to the well-formedness constraint: the variable being substituted can only appear below a \times or \rightarrow node). Because all terms and types are finite, this makes the definition well-founded.

The interpretation of types as parts of \mathcal{D} is then defined as $\llbracket \tau \rrbracket = \{d \mid (d : \tau)\}$. Note that Ω is not part of any type, as expected.

In this framework, we consider XML types as regular tree languages. An XML tree type is interpreted as the set of documents that match the type.

2.4 Subtyping

The subtyping relation is defined as $\tau_1 \leq \tau_2 \Leftrightarrow \llbracket \tau_1 \rrbracket \subset \llbracket \tau_2 \rrbracket$, or, equivalently, $\llbracket \tau_1 \wedge \neg \tau_2 \rrbracket = \emptyset$.

3. Tree logic framework

In this section we introduce the logic in which we model the semantic subtyping framework. This logic is a subset of the one proposed in [10]: a variant of μ -calculus whose models are finite trees. We first introduce below the syntax and semantics of the logic, before tuning it for representing types.

3.1 Formulas

Formulas are defined thus:

$\varphi, \psi ::=$		formula
	\top	true
	$\sigma \mid \neg \sigma$	atomic proposition (negated)
	X	variable
	$\varphi \vee \psi$	disjunction
	$\varphi \wedge \psi$	conjunction
	$\langle a \rangle \varphi \mid \neg \langle a \rangle \top$	existential (negated)
	$\mu(X_i = \varphi_i)_{i \in I} \text{ in } \psi$	(least) n-ary fixpoint

where $a \in \{1, 2\}$ are *programs*, and I is a finite set. Atomic propositions σ correspond to labels from a countable set Σ . Additionally, we use the abbreviation $\mu X. \varphi$ for $\mu(X = \varphi)$ in φ .

3.2 Semantic domain

The semantic domain is the set \mathcal{F} of focused trees defined by the following syntax, where we have an alphabet Σ of labels, ranged over by σ :

$t ::=$	$\sigma[tl]$	tree
$tl ::=$		list of trees
	ϵ	empty list
	$t :: tl$	cons cell
$c ::=$		context
	(tl, Top, tl)	root of the tree
	$(tl, c[\sigma], tl)$	context node
$f ::=$	(t, c)	focused tree

A focused tree (t, c) is a pair consisting of a tree t and its context c . The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be *Top* if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

The *name* of a focused tree is defined as $\text{nm}(\sigma[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions, or *modalities*, that can be followed: for a focused tree f , $f \langle 1 \rangle$ changes the focus to the first child of the current tree, $f \langle 2 \rangle$ changes the focus to the next sibling of the current tree, $f \langle \bar{1} \rangle$ changes the focus to the parent of the tree if *the current tree is a leftmost sibling*, and $f \langle \bar{2} \rangle$ changes the focus to the previous sibling.

Formally, we have:

$$\begin{aligned}
(\sigma[t :: tl], c) \langle 1 \rangle &\stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma], tl)) \\
(t, (tl_l, c[\sigma], t' :: tl_r)) \langle 2 \rangle &\stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma], tl_r)) \\
(t, (\epsilon, c[\sigma], tl)) \langle \bar{1} \rangle &\stackrel{\text{def}}{=} (\sigma[t :: tl], c) \\
(t', (t :: tl_l, c[\sigma], tl_r)) \langle \bar{2} \rangle &\stackrel{\text{def}}{=} (t, (tl_l, c[\sigma], t' :: tl_r))
\end{aligned}$$

When the focused tree does not have the required shape, these operations are not defined.

3.3 Interpretation

Formulas are interpreted as subsets of \mathcal{F} in the following way, where V is a mapping from variables to formulas:

$$\begin{aligned} \llbracket \top \rrbracket_V &\stackrel{\text{def}}{=} \mathcal{F} & \llbracket \sigma \rrbracket_V &\stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) = \sigma\} \\ \llbracket X \rrbracket_V &\stackrel{\text{def}}{=} V(X) & \llbracket \neg\sigma \rrbracket_V &\stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) \neq \sigma\} \\ \llbracket \varphi \vee \psi \rrbracket_V &\stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V \cup \llbracket \psi \rrbracket_V & \llbracket \varphi \wedge \psi \rrbracket_V &\stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V \cap \llbracket \psi \rrbracket_V \\ \llbracket \langle a \rangle \varphi \rrbracket_V &\stackrel{\text{def}}{=} \{f \langle \bar{a} \rangle \mid f \in \llbracket \varphi \rrbracket_V \wedge f \langle \bar{a} \rangle \text{ defined}\} \\ \llbracket \neg \langle a \rangle \top \rrbracket_V &\stackrel{\text{def}}{=} \{f \mid f \langle a \rangle \text{ undefined}\} \end{aligned}$$

$$\begin{aligned} \llbracket \mu(X_i = \varphi_i)_{i \in I} \text{ in } \psi \rrbracket_V &\stackrel{\text{def}}{=} \\ \text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, \llbracket \varphi_j \rrbracket_{V[\overline{T_i/X_i}]} \subset T_j\} &\text{ in} \\ \text{let } (U_j) = (\bigcap_{(T_i) \in S} T_j)_{j \in I} &\text{ in } \llbracket \psi \rrbracket_{V[\overline{U_j/X_i}]} \end{aligned}$$

where $V[\overline{T_i/X_i}(X) = V(X)$ if $X \notin \{X_i\}$ and T_i if $X = X_i$.

The lemma 4.2 of [10] says that the interpretation of a fixpoint formula is equal to the union of the interpretations of all its finite unfoldings (where unfolding is defined as usual). A consequence (detailed in [10]) is that the logic is closed under negation, i. e. for any closed φ , $\neg\varphi$ can be expressed in the syntax using De Morgan's relations and this definition:

$$\begin{aligned} \neg \langle a \rangle \varphi &\stackrel{\text{def}}{=} \neg \langle a \rangle \top \vee \langle a \rangle \neg\varphi \\ \neg \mu(X_i = \varphi_i) \text{ in } \psi &\stackrel{\text{def}}{=} \mu(X_i = \neg\varphi_i \{\overline{X_i/\neg X_i}\}) \text{ in } \neg\psi \{\overline{X_i/\neg X_i}\} \end{aligned}$$

In the following, we consider only closed formulas and write $\llbracket \varphi \rrbracket$ for $\llbracket \varphi \rrbracket_\emptyset$.

4. Logical Encoding

In the context of the present paper, we want finite tree models of the logic to correspond to types introduced in section 2. Thus, we first extend the alphabet of node labels to be able to reason with type constructors. Then, we present the translation of a type into a logical formula.

4.1 Representation of domain elements

Let \mathcal{T} be the set of (unfocused) trees. Set $\mathcal{C} = \{\mathbb{B}[tl] \mid tl \in \mathcal{T}^*\}$, where \mathbb{B} is a label **not in** Σ : the set of trees with a distinguished root \mathbb{B} . Let \mathcal{T}_{ext} be the set of trees obtained by extending Σ with the four extra labels (\rightarrow) , (\times) , \mathbb{B} and Ω . Then \mathcal{D}_Ω can straightforwardly be embedded into \mathcal{T}_{ext} in the following way:

$$\begin{aligned} \text{tree}(c) &= c \\ \text{tree}(\Omega) &= \Omega[\epsilon] \\ \text{tree}(d, d') &= (\times)[\text{tree}(d) :: \text{tree}(d') :: \epsilon] \\ \text{tree}(\{(d_1, d'_1), \dots, (d_n, d'_n)\}) &= \\ (\rightarrow)[\text{tree}(d_1, d'_1) :: \dots :: \text{tree}(d_n, d'_n) :: \epsilon] \end{aligned}$$

In the following we consider this embedding implicitly done, so $\mathcal{D}_\Omega \subset \mathcal{T}_{ext}$.

4.2 Translation of types

First of all, we can define basic types b , which are to represent sets of trees with no special nodes but the distinguished root \mathbb{B} , as the (closed) base formulas of the logic. The full interpretation of formulas uses sets of focused trees, but note that a toplevel formula cannot contain any constraint on what is above or to the left of the node at focus, so it can be considered as describing just a list of

trees. The interpretation of a base type will then be a \mathbb{B} root whose list of children is described by the formula. Formally:

$$\mathbb{B}[\llbracket \varphi \rrbracket] \stackrel{\text{def}}{=} \{\mathbb{B}[t :: tl_2] \mid (t, (tl_1, c[\sigma], tl_2)) \in \llbracket \varphi \rrbracket\}$$

Note how the only part of the context taken into account in defining the semantics is the list of following siblings of the current node.

Then, we translate the types into *extended* formulas obtained (as for extended trees) by adding to Σ the labels (\times) , (\rightarrow) , Ω and \mathbb{B} . Straightforwardly these formulas denote lists of trees in \mathcal{T}_{ext} .

First define the following formulas:

$$\begin{aligned} \text{ibase} &= \mu X.((\neg \langle 1 \rangle \top \vee \langle 1 \rangle X) \wedge (\neg \langle 2 \rangle \top \vee \langle 2 \rangle X) \\ &\quad \wedge \neg \mathbb{B} \wedge \neg (\rightarrow) \wedge \neg (\times) \wedge \neg \Omega) \\ \text{error} &= \Omega \wedge \neg \langle 1 \rangle \top \\ \text{isd} &= \mu X.(\\ &\quad (\mathbb{B} \wedge \langle 1 \rangle \text{ibase}) \vee \\ &\quad ((\times) \wedge \langle 1 \rangle (X \wedge \langle 2 \rangle (X \wedge \neg \langle 2 \rangle \top))) \vee \\ &\quad ((\rightarrow) \wedge \neg \langle 1 \rangle \top \vee \\ &\quad \langle 1 \rangle \mu Y.((\neg \langle 2 \rangle \top \vee \langle 2 \rangle Y) \wedge \\ &\quad (\times) \wedge \langle 1 \rangle (X \wedge \langle 2 \rangle ((X \vee \text{error}) \wedge \neg \langle 2 \rangle \top))) \\ &\quad \text{)))) \end{aligned}$$

ibase selects all tree lists which do not contain any of the special labels (the fixpoint is for selecting all the nodes). *error* is straightforward. *isd* selects all elements of \mathcal{D} (actually, all tree lists whose first element is in \mathcal{D}): either they are a constant (a \mathbb{B} node with a base list as children), or a pair (a (\times) node with exactly two children each of which is itself in \mathcal{D}), or a function: a (\rightarrow) node with either no children at all or a list of children (described by Y) all of which are pairs whose second element may be *error*.

We now associate to every type τ the formula $\text{fullform}(\tau) = \text{isd} \wedge \text{form}(\tau)$, with $\text{form}(\tau)$ defined as follows, where X_v is a different variable for every v and is also different from X :

$$\begin{aligned} \text{form}(b) &= \mathbb{B} \wedge \langle 1 \rangle b \\ \text{form}(\tau_1 \times \tau_2) &= (\times) \wedge \langle 1 \rangle (\text{form}(\tau_1) \wedge \langle 2 \rangle \text{form}(\tau_2)) \\ \text{form}(\tau_1 \rightarrow \tau_2) &= (\rightarrow) \wedge \neg \langle 1 \rangle \top \vee \\ &\quad \langle 1 \rangle \mu X.((\neg \langle 2 \rangle \top \vee \langle 2 \rangle X) \\ &\quad \wedge \langle 1 \rangle (\neg \text{form}(\tau_1) \vee \langle 2 \rangle \text{form}(\tau_2))) \\ &\quad) \\ \text{form}(\tau_1 \vee \tau_2) &= \text{form}(\tau_1) \vee \text{form}(\tau_2) \\ \text{form}(\neg\tau) &= \neg \text{form}(\tau) \\ \text{form}(\mathbf{0}) &= \neg \top \\ \text{form}(v) &= X_v \\ \text{form}(\mu v. \tau) &= \mu X_v. \text{form}(\tau) \end{aligned}$$

Recall that basic types b are themselves formulas, but that their interpretation as a type is different from their interpretation as a formula (see the first paragraph of Section 4.2 and the definition of $\mathbb{B}[\llbracket \varphi \rrbracket]$, the interpretation as a type, in terms of $\llbracket \varphi \rrbracket$, the interpretation as a formula). This explains why the translation of b contains b itself. The translation of product types is simple: it describes a (\times) node whose first child is described by $\text{form}(\tau_1)$ and has a following sibling described by $\text{form}(\tau_2)$. The translation of arrow types has a structure similar to what appeared in *isd*: it describes a (\rightarrow) node with either no children or a list of children recursively described by X (each node has either no following sibling or a following sibling itself described by X). Each of these nodes must have a first child which either is not of type τ_1 or has a next sibling of type τ_2 — this means that these nodes represent pairs (d_i, d'_i) such that

$(d_i : \tau_1) \Rightarrow (d'_i : \tau_2)$. The attentive reader may notice that the formula $\text{form}(\tau_1 \rightarrow \tau_2)$ does not enforce in itself that all children of the (\rightarrow) node are actually pairs; the reason for that is that isd already enforces it.

We can see that the formulas in the translation do not contain any $\langle 2 \rangle$ at toplevel (i. e. not under $\langle 1 \rangle$), nor does isd . This means they describe a single tree (they say nothing on its siblings), or in other words that in their interpretation as focused trees, the context is completely arbitrary, as it is not constrained in any way. Formally, we thus define the restricted interpretation of extended formulas as follows:

$$\mathbb{F}[\varphi] \stackrel{\text{def}}{=} \{t \mid (t, c) \in \llbracket \varphi \rrbracket\}$$

That is, we drop the context completely.

Then we have $\mathbb{F}[\text{fullform}(\tau)] = \llbracket \tau \rrbracket$. This is a particular case of the property for polymorphic types which will be proved in the following section.

The main consequence of this property is that a type τ is empty if and only if the interpretation of the corresponding formula is empty — which is equivalent to the formula being unsatisfiable. Because there exists a satisfiability-checking algorithm for this tree logic [10], this means this translation gives an alternative way to decide the classical semantic subtyping relation as defined in [9]. More interestingly, it yields a decision procedure for the subtyping relation *in the polymorphic case as well*, as we will explain in the next section.

5. Polymorphism: Supporting Type Variables

So far we have described a new, logic-based approach to a question — semantic subtyping in the presence of intersection, negation and arrow types — which had already been studied. We now show how this new approach allows us, in a very natural way, to encompass the latest work by adding polymorphism to the types along the lines of [5].

We add to the syntax of types *variables*, α, β, γ taken from a countable set \mathcal{V} . If τ is a polymorphic type, we write $\text{var}(\tau)$ the set of variables it contains and call *ground type* a type with no variable. We sometimes write $\tau(\bar{\alpha})$ to indicate that $\text{var}(\tau)$ is included in $\bar{\alpha}$.

5.1 Subtyping in the polymorphic case: a problem of definition

The intuition of subtyping in the presence of type variables is that $\tau_1(\bar{\alpha}) \leq \tau_2(\bar{\alpha})$ should hold true whenever, *independently of the variables* $\bar{\alpha}$, any value of type τ_1 has type τ_2 as well. However the correct definition of ‘independently’ is not obvious. It should look like this:

$$\forall \bar{\alpha}, \llbracket \tau_1(\bar{\alpha}) \rrbracket \subset \llbracket \tau_2(\bar{\alpha}) \rrbracket$$

but because variables are abstractions, it is not completely clear over what to quantify them. As mentioned in [13], a candidate — naive — definition would use *ground substitutions*, that is, if the inclusion of interpretations always holds when variables are replaced with ground types, then the subtyping relation holds:

$$\tau_1(\bar{\alpha}) \leq \tau_2(\bar{\alpha}) \Leftrightarrow \forall \bar{\tau} \text{ ground types, } \llbracket \tau_1(\bar{\tau}/\bar{\alpha}) \rrbracket \subset \llbracket \tau_2(\bar{\tau}/\bar{\alpha}) \rrbracket \quad (2)$$

Obviously the condition on the right must be *necessary* for subtyping to hold. But deciding that it is *sufficient* as well makes the relation unsatisfactory and somehow counterintuitive, as remarked in [13]. Indeed, suppose int is an *indivisible* type, that is, that it has no subtype beside $\mathbf{0}$ and itself. Then the following would hold:

$$\text{int} \times \alpha \leq (\text{int} \times \neg\text{int}) \vee (\alpha \times \text{int}) \quad (3)$$

This relation abuses the definition by taking advantage of the fact that for any ground type τ , either $\llbracket \text{int} \rrbracket \subset \llbracket \tau \rrbracket$ or $\llbracket \tau \rrbracket \subset \llbracket \neg\text{int} \rrbracket$. In the first case, because $\llbracket \tau \rrbracket \subset (\llbracket \neg\text{int} \rrbracket \cup \llbracket \text{int} \rrbracket)$, we have $\llbracket \text{int} \times \tau \rrbracket \subset$

$\llbracket \text{int} \times \neg\text{int} \rrbracket \cup \llbracket \text{int} \times \text{int} \rrbracket$ and then the second member of the union is included in $\llbracket \tau \times \text{int} \rrbracket$. In the second case, we directly have $\llbracket \text{int} \times \tau \rrbracket \subset \llbracket \text{int} \times \neg\text{int} \rrbracket$.

This trick, which only works with indivisible ground types, not only shows that candidate definition (2) yields bizarre relations where a variable occurs in unrelated positions on both sides. It also means the candidate definition is very sensitive to the precise semantics of base types, since it distinguishes indivisible types from others. More precisely, it means that refining the collection of base types, for example by adding types even and odd, can break subtyping relations which held true without these new types — this is simply due to the fact that it increases the set over which $\bar{\tau}$ is quantified in (2), making the relation stricter. This could hardly be considered a nice feature of the subtyping relation.

The conclusion is thus that the types in (3) should be considered related by *chance* rather than by necessity, hence not in the subtyping relation, and that quantifying over all possible ground types is not enough; in other words, candidate definition (2) is too weak and does not properly reflect the intuition of ‘independently of the variables’. Indeed, (3) is in fact dependent on the variable as we saw, the point being that there are only two cases and that the convoluted right-hand type is crafted so that the relation holds in both of them, though for different reasons.

In order to restrict the definition of subtyping, [13], which concentrates on XML types, uses a notion of *marking*: some parts of a value can be marked (using paths) as corresponding to a variable, and the relation ‘a value has a type’ is changed into ‘a marked value matches a type’, so the semantics of a type is not a set of values but of pairs of a value and a marking. This is designed so that it integrates well in the XDuce language, which has pattern-matching but no higher-order functions (hence no arrow types), so their system is tied to the operational semantics of matching and provides only a partial solution.

The question of finding the correct definition of semantic subtyping in the polymorphic case was finally settled very recently by Castagna and Xu [5]. Their definition does, in the same way as (2), follow the idea of a universal quantification over possible *meanings* of variables but solves the problem raised by (3) by using a much larger set of possible meanings — thus yielding a stricter relation. More precisely, variables are allowed to represent not just ground types but any arbitrary part of the semantic domain; furthermore, the semantic domain itself must be large enough, which is embodied by the notion of *convexity*. We refer the reader to [5] for a detailed discussion of this property and its relation to the notion of *parametricity* studied by Reynolds in [14]; we will here limit ourselves to introducing the definitions strictly necessary for the discussion at hand.

In this work, we do not use this definition with its universal quantification directly. Rather, we retain from [13] the idea of tagging (pieces of) values which correspond to variables, but do so in a more abstract way, by extending the semantic domain, and define a *fixed* interpretation of polymorphic types in this extended domain as a straightforward extension of the monomorphic framework. We then show how to build a set-theoretic model of polymorphic types, in the sense of [5], based on this domain, and prove that the inclusion relation on fixed interpretations is equivalent to the full subtyping relation induced by this model. Finally, we explain briefly the notion of convexity and show that this model is convex, implying that this relation is, in fact, the semantic subtyping relation on polymorphic types, as defined in [5]. These steps are formally detailed in the following section.

5.2 Interpretation of polymorphic types

Let Λ be an infinite set of optional labels, and ι an injective function from \mathcal{V} to Λ . (It would be possible to set $\Lambda = \mathcal{V}$, but for clarity

we prefer to distinguish *labels* which tag elements of the semantic domain from *variables* which occur in types.) We extend the grammar of (extended) trees by allowing any node to bear, in addition to its single σ label from $\Sigma \cup \{(\rightarrow), (\times), \mathbb{B}, \Omega\}$, any (finite) number of labels from Λ . We write it $\sigma_L[tl]$ where L is a finite part of Λ . We extend \mathcal{C} and \mathcal{D} accordingly. When using the non-tree form of types, for instance (d_1, d_2) , we indicate the set of root labels on the bottom right like this: $(d_1, d_2)_L$ (here L is the set of labels borne by the (\times) node constituting the root of the pair tree).

We then extend the predicate defining the interpretation of types given in Section 2.3 with the following additional case:

$$(\sigma_L[tl] : \alpha) = \iota(\alpha) \in L$$

In other words, the interpretation of a type variable is the set of all trees whose root bears the label corresponding to that variable. The other cases are unchanged, except that the semantic domain is now much larger. This means that the same definition leads to larger interpretations; in particular, the interpretation of a (nonempty) ground type is always an infinite set which contains all possible labellings for each of its trees.

Subtyping over polymorphic types is then defined, as before, as set inclusion between interpretations:

$$\tau_1(\bar{\alpha}) \leq \tau_2(\bar{\alpha}) \Leftrightarrow \llbracket \tau_1(\bar{\alpha}) \rrbracket \subset \llbracket \tau_2(\bar{\alpha}) \rrbracket \quad (4)$$

It may seem strange to give type *variables* a *fixed* interpretation, and on the other hand it may seem surprising that this definition of subtyping does not actually contain any quantification and is nevertheless stronger than (2) which contains one. The keypoint is that a form of universal quantification is implicit in the extension of the semantic domain: in some sense, the interpretation of a variable represents all possible values of the variable *at once*. Indeed, for any variable α and any tree d in the domain, there always exist both an infinity of copies of d which are in the interpretation of α and another infinity of copies which are not. From the point of view of logical satisfiability, this makes the domain big enough to contain all possible cases.

In order to show that, despite the appearances, Definition (4) accurately represents a relation that holds *independently of the variables*, we rely, as discussed above, on the formal framework developed by Castagna and Xu [5]. For this, we first introduce *assignments* η : functions from \mathcal{V} to $\mathcal{P}(\mathcal{D})$ (where \mathcal{D} is the extended semantic domain with labels). Thus an assignment attributes to each variable an arbitrary set of elements from the semantic domain.

We then define the interpretation of a type *relative to an assignment* in the following way: the predicate $(d' :_{\eta} \tau)$ is defined inductively in the same way as the $(d' : \tau)$ of Section 2.3 but with the additional clause:

$$(d :_{\eta} \alpha) = d \in \eta(\alpha).$$

The interpretation of the polymorphic type τ relative to the assignment η is then $\llbracket \tau \rrbracket_{\eta} = \{d \mid (d :_{\eta} \tau)\}$. This defines an infinity of possible interpretations for a type, depending on the actual values assigned to the variables, and constitutes a set-theoretic model of types in the sense of [5]. The subtyping relation induced by this model is the following:

$$\tau_1(\bar{\alpha}) \leq \tau_2(\bar{\alpha}) \Leftrightarrow \forall \eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}, \llbracket \tau_1(\bar{\alpha}) \rrbracket_{\eta} \subset \llbracket \tau_2(\bar{\alpha}) \rrbracket_{\eta} \quad (5)$$

which we can more easily compare to the candidate definition (2): it does in the same way quantify over possible meanings of the variables but uses a much larger set of possible meanings, yielding a stricter relation. We will now prove that this relation is, for our particular model, actually equivalent to (4).

For this, let us first define the *canonical assignment* η_L as follows:

$$\eta_L(\alpha) \stackrel{\text{def}}{=} \{\sigma_L[tl] \in \mathcal{D} \mid \iota(\alpha) \in L\}.$$

Then it is easily seen that the fixed interpretation $\llbracket \tau \rrbracket$ of a polymorphic type is the same as its interpretation relative to the canonical assignment, $\llbracket \tau \rrbracket_{\eta_L}$. What we would like to prove is that the canonical assignment is somehow representative of all possible assignments, making the fixed interpretation sufficient for the purpose of defining subtyping. This is done by the following lemma and corollary.

LEMMA 5.1. *Let V be a finite part of \mathcal{V} . Let η be an assignment. Let T be the set of all types τ such that $\text{var}(\tau) \subset V$. Then there exists a function $F_V^{\eta} : \mathcal{D} \rightarrow \mathcal{D}$ such that: $\forall \tau \in T, \forall d \in \mathcal{D}, d \in \llbracket \tau \rrbracket_{\eta} \Leftrightarrow F_V^{\eta}(d) \in \llbracket \tau \rrbracket_{\eta_L}$.*

Proof: For d in \mathcal{D} , let $L(d) = \{\iota(\alpha) \mid \alpha \in V \wedge d \in \eta(\alpha)\}$. Since V is finite, $L(d)$ is finite as well. We define $F_V^{\eta}(d)$ inductively as follows:

- if $d = \mathbb{B}_L[tl]$ then $F_V^{\eta}(d) = \mathbb{B}_{L(d)}[tl]$
- if $d = (d_1, d_2)_L$ then $F_V^{\eta}(d) = (F_V^{\eta}(d_1), F_V^{\eta}(d_2))_{L(d)}$
- $F_V^{\eta}(\Omega) = \Omega$
- if $d = \{(d_1, d'_1), \dots, (d_n, d'_n)\}_L$ then $F_V^{\eta}(d) = \{(F_V^{\eta}(d_1), F_V^{\eta}(d'_1)), \dots, (F_V^{\eta}(d_n), F_V^{\eta}(d'_n))\}_{L(d)}$

So F_V^{η} preserves the structure but changes the labels so that the root node of $F_V^{\eta}(d)$ is labelled with $L(d)$ and so on inductively for its subterms.

Let $\mathcal{P}(d, \tau) = d \in \llbracket \tau \rrbracket_{\eta} \Leftrightarrow F_V^{\eta}(d) \in \llbracket \tau \rrbracket_{\eta_L}$. We prove that it holds for all pairs (d, τ) such that τ is in T by induction on those pairs, using the ordering relation on them defined in Section 2.3, noticing that $\tau \in T$ implies that all subterms (and unfoldings) of τ are in T as well. The base cases are:

- if τ is a variable. Then it is in V by hypothesis and $\mathcal{P}(d, \tau)$ is true by definition of $L(d)$.
- if it is a base type. Then $\mathcal{P}(d, \tau)$ is true because the interpretation of τ is independent of assignments and labellings.

For the inductive cases, we suppose the property true for all strictly smaller pairs (d, τ) such that τ is in T .

- For the arrow and product cases, the inductive definition of F_V^{η} makes the result straightforward.
- For the negation and disjunction cases, the result is immediate from the induction hypothesis.
- For $\mu\nu.\tau$, recall that the well-formedness constraint on types implies that the type's unfolding has a strictly smaller shallow depth than the original type, hence we can use the induction hypothesis on the unfolding and conclude.

COROLLARY 5.2. *Let τ be a type. $\bigcup_{\eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}} \llbracket \tau \rrbracket_{\eta} = \emptyset$ if and only if $\llbracket \tau \rrbracket_{\eta_L} = \emptyset$.*

Proof: If the union is not empty, there exists η and d such that $d \in \llbracket \tau \rrbracket_{\eta}$. From the previous lemma we then have $F_{\text{var}(\tau)}^{\eta}(d) \in \llbracket \tau \rrbracket_{\eta_L}$.

This corollary shows that the canonical assignment is representative of all possible assignments and implies that the subtyping relation defined by (4) is equivalent to the one defined by (5).

Convexity of the model. Definition (5) corresponds to semantic subtyping as defined in [5], but only on the condition that the underlying model of types be *convex*. Indeed, we can see that this definition is dependent on the set of possible assignments, which itself depends on the chosen (abstract) semantic domain, so it is reasonable to think that increasing the semantic domain could restrict the relation further. In other words, for the definition to be correct, the domain must be large enough to cover all cases. Castagna and Xu's *convexity* characterises this notion of 'large

enough'. The property is the following: a set-theoretic model of types is *convex* if, whenever a finite collection of types τ_1 to τ_n each possess a nonempty interpretation relative to some assignment, then there exists a common assignment making all interpretations nonempty at once. This reflects the idea that there are enough elements in the domain to witness all the cases.

In our case, it comes as no surprise that the extended model of types is convex since any nonempty ground type has an infinite interpretation, which, as proved in [5], is a sufficient condition. But we need not even rely on this result since Corollary 5.2 proves a property even stronger than convexity: having a nonempty interpretation relative to *some* assignment is the same as having a nonempty interpretation relative to *the* common canonical assignment. This stronger property makes the apparently weaker relation defined by (4) equivalent, in our particular model, to the full semantic subtyping relation Castagna and Xu defined. This allows us to reduce the problem of deciding their relation to a question of inclusion between fixed interpretations, making the addition of polymorphism a mostly straightforward extension to the logical encoding we presented for the monomorphic case.

Interestingly, in [5] the authors suggest that convexity constrains the relation enough that it should allow reasoning on types, similarly to the way parametricity allowed Wadler [16] to deduce ‘theorems for free’ from typing information. The fact that our logical reasoning approach very naturally has this convexity property — indeed, it is difficult to think of a logical representation of variables which would not have it — seems to corroborate their intuition, although reasoning on types beyond deciding subtyping is currently left as future work.

We now show how this extension of the type system is encoded in our logic.

5.3 Logical encoding of variables

We extend the logic with atomic propositions α which behave similarly as σ except they are not mutually exclusive. The interpretation of these propositions is defined as:

$$\llbracket \alpha \rrbracket = \{(\sigma_L[t], c) \mid \iota(\alpha) \in L\}$$

$$\llbracket \neg\alpha \rrbracket = \{(\sigma_L[t], c) \mid \iota(\alpha) \notin L\}$$

The translation $\text{form}(\tau)$ of types into formulas is extended in the obvious way by $\text{form}(\alpha) = \alpha$.

THEOREM 5.3. *With these extended definitions, $\mathbb{F}[\text{fullform}(\tau)] = \llbracket \tau \rrbracket$.*

Proof (sketch): Preliminary remark: whenever φ does not contain any $\langle 2 \rangle$ at toplevel (which is the case of the formulas representing types), then $\llbracket \varphi \rrbracket = \mathbb{F}[\varphi] \times \mathbf{C}$ where \mathbf{C} is the set of all possible contexts. Hence, when considering such formulas, set-theoretic relations between full interpretations are equivalent to the same relations between first components.

First we check that $\mathbb{F}[\text{isd}] = \mathcal{D}$ and reformulate the statement as $\mathcal{D} \cap \mathbb{F}[\text{form}(\tau)] = \llbracket \tau \rrbracket$.

We make the embedding function tree explicit for greater clarity. What we have to show is that, for any d in \mathcal{D} , we have $(d : \tau)$ if and only if $(\text{tree}(d), c)$ is in $\llbracket \text{form}(\tau) \rrbracket$ for some (or, equivalently, for any) c .

The property is proved by induction on the pair (d, τ) , following the definition of the predicate:

- for $(c : b)$ it holds by definition.
- for $((d_1, d_2)_L : \tau_1 \times \tau_2)$, let $f = (\text{tree}((d_1, d_2)_L), c)$. f is in $\llbracket \text{form}(\tau_1 \times \tau_2) \rrbracket$ if and only if $f \langle 1 \rangle$ is in $\llbracket \text{form}(\tau_1) \rrbracket$ and $f \langle 1 \rangle \langle 2 \rangle$ is in $\llbracket \text{form}(\tau_2) \rrbracket$. (We already know that the node name is (\times) by the structure of d .) Just see that the tree rooted at $f \langle 1 \rangle$ is $\text{tree}(d_1)$ and the one at $f \langle 1 \rangle \langle 2 \rangle$ is $\text{tree}(d_2)$.

- for functions, use the finite unfolding property and the fact the set of pairs is finite, then see, similarly as above, that the correct properties are enforced when navigating the tree.
- for union, negation and empty types, use the preliminary remark.
- for $(d : \alpha)$, just see that $d \in \iota(\alpha)$ and $d \in \mathbb{F}[\alpha]$ both mean that the root node of d , which is the node at focus in the formula, bears the label $\iota(\alpha)$.
- for $(d : \mu\nu.\tau)$, use the property that the interpretation of a fixpoint formula and its unfolding are the same (lemma 4.2 of [10]).

COROLLARY 5.4. $\tau_1 \leq \tau_2$ holds if and only if $\text{fullform}(\tau_1 \wedge \neg\tau_2)$, or alternatively $\text{isd} \wedge \text{form}(\tau_1) \wedge \neg\text{form}(\tau_2)$, is unsatisfiable.

5.4 Complexity

LEMMA 5.5. *Provided two types τ_1 and τ_2 , the subtyping relation $\tau_1 \leq \tau_2$ can be decided in time $2^{\mathcal{O}(|\tau_1|+|\tau_2|)}$ where $|\tau_i|$ is the size of τ_i .*

Proof (sketch): The logical translation of types performed by the function $\text{form}(\cdot)$ does not involve duplication of subformulas of variable size, therefore $\text{form}(\tau)$ is of linear size with respect to $|\tau|$. Since isd has constant size, the whole translation $\text{fullform}(\tau)$ is linear in terms of $|\tau|$. For testing satisfiability of the logical formula, we use the satisfiability-checking algorithm presented in [10] whose time complexity is $2^{\mathcal{O}(n)}$ in terms of the formula size n .

6. Practical Experiments

In this section we report on some interesting lessons learned from practical experiments with the implementation of the system in order to prove relations in the type algebra.

6.1 Implementation

The algorithm for deciding the subtyping relation has been fully implemented on top of the satisfiability solver introduced in [10]. Our implementation is publicly available. Interaction with the system is offered through a user interface in a web browser. The system is available online at:

<http://wam.inrialpes.fr/websolver/>

A screenshot of the interface is given in Figure 1. The user can either enter a formula through area (1) of Figure 1 or select from pre-loaded analysis tasks offered in area (4) of Figure 1. The level of details displayed by the solver can be adjusted in area (2) of Figure 1 and makes it possible to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 1 together with counter-examples.

In the polymorphic setting, a counter-example, that is, a model satisfying a formula, is in principle, according to the extended semantics, a labelled tree. However, as mentioned in Section 5.2, whenever a formula is satisfiable there always exists an infinity of possible labellings which satisfy it. Therefore, rather than proposing just one labelled tree, the solver gives a minimal tree together with *labelling constraints* representing all labellings which make that particular tree a counter-example. Namely, for each variable α , every node will be labelled with α to indicate that it *must* be labelled with α for the formula to be satisfied, with $\neg\alpha$ to indicate that it *must not* be, or with nothing if label α is irrelevant for that particular node. This allows an easier interpretation of the counter-example in terms of assignments: the subtyping relation fails whenever the assignment for each variable α contains all the trees whose root is marked with α and none of those whose root is marked with $\neg\alpha$.

XML Reasoning Solver Project

Home
Demo
Documentation
Publications
Team

Enter your formula below:

```
nsubtype (¬_a → _b, ((T → F) → _b) | _a)
```

See [user manual](#) or pick an example

- [XPath Satisfiability #1](#)
- [XPath Satisfiability #2](#)
- [XPath Containment](#)
- [XPath Equivalence](#)
- [Mu-formula with values](#)
- [Mu-formula with recursion](#)
- [Polymorphism with arrow types #1](#)
- [Polymorphism with arrow types #2](#)

Advanced Options **Check Satisfiability** Execution completed.

XML Attributes

Show Lean

Show Formula

Formula Statistics

Input parsed and compiled [total time: 2 ms].

Global formula contains the following total numbers of occurrences (including duplicates):
16 atomic propositions, 36 modalities, 15 variables, 5 fixpoint binder, 22 negations, 22 conjunctions, 18 disjunctions.

Satisfiability Tested Formula:

```
(mu X11.((
  (let_mu
    X6=((BASE & <1>(mu X5.(((¬
      (<1>T) | <1>X5) & (¬<2>T) | <2>X5)) & (¬(ERROR) & ¬(BASE) & ¬(FUNCTION) & ¬(PAIR)))) | (PAIR & <1>(X6 & <2>(X6 & ¬<2>T)))) | (FUNCTION & (¬<1>T) | <1>X7))),
    in
    X6) & (FUNCTION & (¬<1>T) | <1>(mu X1.((¬
      (<2>T) | <2>X1) & <1>_a | <2>_b)))))) & ((¬(FUNCTION) | <1>T & (¬<1>T) | <1>(mu X9.((¬
      (<1>T) | <1>(FUNCTION & (¬<1>T) | <1>T & (¬<1>T) | <1>(mu X8.F)))))) & (¬<2>T) | <2>¬_b)))) | ((<2>X9 | ¬<2>T) & <2>T)))) & ¬_a)) | <1>X11 | <2>X11)))
```

Computing Relevant Closure...
Computed Relevant Closure [4 ms].
Computed Lean [0 ms].
Lean size is 30. It contains 23 eventualities and 7 symbols.
Computing Fixpoint...[9 ms].

Formula is satisfiable [total time: 17 ms].
A satisfying finite binary tree model is [5 ms]:
FUNCTION ¬_a(PAIR(FUNCTION _a(#, ERROR ¬_b), #), #)
In XML syntax:
<FUNCTION ¬_a xmlns:solver="http://wam.inrialpes.fr/xml" solver:target="true">
<PAIR>
<FUNCTION _a/>
<ERROR ¬_b/>
</PAIR>
</FUNCTION>

This online demo is a 100% Java implementation of the solver that runs inside a Tomcat servlet. It is based on a thread-safe re-implementation of a BDD package (JavaBDD). However, the performance of this package is very slow compared to what can be achieved with an off-line solver implementation with native BDDs. Ask us if you are interested in the high-speed off-line version of the solver.

Figure 1. Screenshot of the Interface.

6.2 Concrete Syntax for Type Algebra

All the examples in the subsection that follows can be tested in our online prototype. For this purpose, the following table gives the correspondence between the syntax used in the paper and the syntax that must be used in the implementation. Additionally, the embedding of a base formula of the logic into a base type is provided by curly braces: $\{\varphi\}$ is an abbreviation for $\text{isbase} \wedge \langle 1 \rangle \varphi$.

	Paper Syntax	Implementation Syntax
Type variables	α, β, γ	$_a, _b, _g$
Type constructors	\times, \rightarrow	$*, \rightarrow$
Recursive types	$\mu\nu.\tau$	$\text{let } \$v = t \text{ in } \v
Basic types	$0, 1$	F, T
Logical connectives	$\wedge, \vee, \neg, \Rightarrow$	$\&, , \sim, \Rightarrow$
Subtyping	$\neg(\tau_1 \leq \tau_2)$	$\text{nsubtype}(t1, t2)$

6.3 Examples and Discussion

The goal of this subsection is to illustrate through some examples how our logical setting is natural and intuitive for proving subtyping relations. For example, one can prove simple properties such as the one below:

$$(\alpha \rightarrow \gamma) \wedge (\beta \rightarrow \gamma) \leq (\alpha \vee \beta) \rightarrow \gamma \quad (6)$$

This is formulated as follows:

$\text{nsubtype}((_a \rightarrow _g) \& (_b \rightarrow _g), (_a | _b) \rightarrow _g)$

which is automatically compiled into the logical formula shown on Figure 2 and given to the satisfiability solver that returns:

Formula is unsatisfiable [16 ms].

which means that no satisfying tree was found for the formula, or, in other terms, that the negation of the formula is valid. The

```

(mu X8.(((
  (let_mu
    X5=(((BASE <1>(mu X4.(((~(<1>T) | <1>X4) & (~(<2>T) | <2>X4))
      & (~(ERROR) & ~(BASE) & ~(FUNCTION) & ~(PAIR))))))
    | (PAIR & <1>(X5 & <2>(X5 & ~(<2>T)))))) | (FUNCTION & (~(<1>T) | <1>X6))),
    X6=(((~(<2>T) | <2>X6) & PAIR) & <1>(X5 & <2>(X5 | (ERROR & ~(<1>T))) & ~(<2>T))))
  in
    X5) & ((FUNCTION & (~(<1>T) | <1>(mu X1.(((~(<2>T) | <2>X1) & <1>(~(_a) | <2>_g))))))
      & (FUNCTION & (~(<1>T) | <1>(mu X2.(((~(<2>T) | <2>X2) & <1>(~(_b) | <2>_g)))))))
      & (~(FUNCTION) | (<1>T & (~(<1>T) | <1>(mu X7.(((~(<2>T) | <2>X7)
        | (~(<1>T) | <1>(~(_a) | _b) & (~(<2>T) | <2>~(_g)))))))))) | (<1>X8 | <2>X8)))

```

Figure 2. Logical translation tested for satisfiability.

satisfiability solver is seen as a theorem prover since its run built a formal proof that property (6) holds.

Jérôme Vouillon [15] uses simple examples with lists to illustrate polymorphism with recursive types. For instance, consider the type of lists of elements of type α :

$$\tau_{\text{list}} = \mu v.(\alpha \times v) \vee \text{nil}$$

where “nil” is a singleton type. The type of lists of an even number of such elements can be written as:

$$\tau_{\text{even}} = \mu v.(\alpha \times (\alpha \times v)) \vee \text{nil}$$

By giving the following formula to the solver :

```

nsubtype(let $v = (_a * _a * $v) | {nil} in $v,
  let $w = (_a * $w) | {nil} in $w )

```

which is found unsatisfiable, we prove that

$$\tau_{\text{even}} \leq \tau_{\text{list}}$$

If we now consider the type of lists of an odd number of elements of type α :

$$\tau_{\text{odd}} = \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil})$$

we can check additional properties in a similar manner, like:

$$(\tau_{\text{even}} \vee \tau_{\text{odd}} \leq \tau_{\text{list}}) \wedge (\tau_{\text{list}} \leq \tau_{\text{even}} \vee \tau_{\text{odd}})$$

The following formula corresponds to the example (1) of the introduction:

```

bool() = {true|false};
list() = let $l = (_a * $l) | {nil} in $l;
odd() = let $o = (_a * _a * $o) | (_a * {nil}) in $o;
even() = let $e = (_a * _a * $e) | {nil} in $e;

```

```

nsubtype ( (odd() -> {true}) & (even() -> {false}),
  list() -> bool() )

```

This formula is found unsatisfiable by the solver, which proves the validity of the subtyping statement (1).

Giuseppe Castagna [see section 2.7 of [5]] gives some examples of non-trivial relations that hold in the type algebra. For instance, the reader can check that the types $\mathbf{1} \rightarrow \mathbf{0}$ and $\mathbf{0} \rightarrow \mathbf{1}$ can be seen as extrema among the function types:

$$\mathbf{1} \rightarrow \mathbf{0} \leq \alpha \rightarrow \beta \quad \text{and} \quad \alpha \rightarrow \beta \leq \mathbf{0} \rightarrow \mathbf{1}$$

Our system also permitted to detect an error in [5] and provided some helpful information to the authors of [5] in order to find the origin of the error and make corrections. Specifically, in a former version of [5], the following relation was considered:

$$(\neg\alpha \rightarrow \beta) \leq ((\mathbf{1} \rightarrow \mathbf{0}) \rightarrow \beta) \vee \alpha \quad (7)$$

Authors explained how this relation was proved by their algorithm. However, by encoding the relation in our system we found that this relation actually does not hold. Specifically, this is formulated as follows in our system:

```

nsubtype (~_a -> _b, ((T -> F) -> _b) | _a)

```

The satisfiability solver, when fed this formula, returns the following counter-example:

```

FUNCTION ~_a (PAIR(FUNCTION _a (#, ~_b ERROR), #), #)

```

FUNCTION represents (\rightarrow) and PAIR represents (\times) . This is a binary tree representation of the n -ary tree

$$(\rightarrow)_{-\alpha}[(\times)[(\rightarrow)_{\alpha}[\epsilon] :: \Omega :: \epsilon] :: \epsilon]$$

which corresponds to the domain element

$$\{(\{\alpha, \Omega\})_{-\alpha}.$$

The inner (\rightarrow) node has no children and thus represents the function which always diverges: $\{\}$. More precisely, it represents a copy f of this function that belongs to the interpretation of α . The root (\rightarrow) node then represents a function which is *not* in $\llbracket \alpha \rrbracket$ and which to f associates an error, while diverging on any other input.

Now, why is it a counter-example to (7)? As the function diverges but on one input f and that input is in $\llbracket \alpha \rrbracket$, it is vacuously true that on all inputs in $\llbracket \neg\alpha \rrbracket$ for which it returns a result, this result is in $\llbracket \beta \rrbracket$. Thus it does have the type on the left-hand side. However, it does not have type α , nor does it have type $((\mathbf{1} \rightarrow \mathbf{0}) \rightarrow \beta)$. Indeed, f does have type $\mathbf{1} \rightarrow \mathbf{0}$ and our counter-example function associates to it an error, which is not in $\llbracket \beta \rrbracket$.

7. Related Work

We review below related works while recalling how the introduction of XML progressively renewed the interests in parametric polymorphism.

The seminal work by Hosoya, Vouillon and Pierce on a type system for XML [12] applied the theory of regular expression types and finite tree automata in the context of XML. The resulting language XDuce [11] is a strongly typed language featuring recursive, product, intersection, union, and complement types. The subtyping relation is decided through a reduction to containment of finite tree automata, which is known to be in EXPTIME. This work does not support function types nor polymorphism, but provided a ground for further research.

In particular, Frisch, Castagna and Benzaken provide a gentle introduction to semantic subtyping in [9]. Semantic subtyping focuses on a set-theoretic interpretation, as opposed to traditional subtyping through direct syntactic rules. Our logical modeling presented in Section 4 naturally follows the semantic subtyping approach as the underlying logic has a set-theoretic semantics. Frisch, Castagna and Benzaken added function types to the semantic subtyping performed by XDuce’s type system. This notably resulted in the CDuce language [2]. However, CDuce does not support type variables and thus lacks polymorphism.

Vouillon studied polymorphism in the context of regular types with arrow types in [15]. Specifically, he introduced a pattern algebra and a subtyping relation defined by a set of syntactic inference

rules. A semantic interpretation of subtyping is given by ground substitution of variables in patterns. The type algebra has the union connective but lacks negation and intersection. The resulting type system is thus less general than ours.

Polymorphism was also the focus of the later work found in [13]. In [5], it is explained that at that time a semantically defined polymorphic subtyping looked out of reach, even in the restrictive setting of [11], which did not account for higher-order functions. This is why [13] fell back on a somewhat syntactic approach linked to pattern-matching that seemed difficult to extend to higher-order functions. Our work shows however that such an extension was possible using similar basic ideas, only slightly more abstract.

The most closely related work is the one found in [5], in the same proceedings as the current paper, which solves the problem of defining subtyping semantically in the polymorphic case for the first time, and addresses the problem of its decision through an ad-hoc and multi-step algorithm, which was only recently proved to terminate in all cases. Our approach also addresses the problem of deciding their subtyping relation and solves it through a more direct, generic, natural and extensible approach since our solution relies on a modeling into a well-known modal logic (the μ -calculus) and on using a satisfiability solver such as the one proposed in [10]. This logical connection also opens the way for extending polymorphic types with several features found in modal logics.

The work of [3] follows the same spirit than ours: typechecking is subcontracted to an external logical solver. An SMT-solver is used to extend a type-checker for the language Dminor (a core dialect for M) with refinement type and type-tests. The type-checking relies on a semantic subtyping interpretation but neither function types nor polymorphism are considered. Therefore, their work is incomparable to ours.

The present work heavily relies on the work presented in [10] since we repurpose the satisfiability-checking algorithm of [10] for deciding the subtyping relation. The goal pursued in [10] was very different in spirit: the goal was to decide containment of XPath queries in the presence of regular tree types. To this end, the decidability of a logic with converse for finite ordered trees is proved in a time complexity which is a simple exponential of the size of the formula. The present work builds on these results for solving semantic subtyping in the polymorphic case.

8. Conclusion

The main contribution of this paper is to define a logical encoding of the subtyping relation defined in [5], yielding a decision algorithm for it. We prove that this relation is decidable with an upper-bound time complexity of $2^{(n)}$, where n is the size of types being checked. In addition, we provide an effective implementation of the decision procedure that works well in practice.

This work illustrates a tight integration between a functional language type-checker and a logical solver. The type-checker uses the logical solver for deciding subtyping, which in turn provides counter-examples (whenever subtyping does not hold) to the type-checker. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold. As a result, our solver represents a very attractive back-end for functional programming languages type-checkers.

This result pushes the integration between programming languages and logical solvers to an advanced level. The proposed logical approach is not only capable of modeling higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logics such as XML tree types. This shows that such logical solvers can become the core of XML-centric functional languages type-checkers such as those used in CDuce or XDuce.

Acknowledgments. We are thankful to Giuseppe Castagna for bringing our attention to the problem of subtyping with arrow types in the polymorphic case. He gave us precious insights for the precise formulation of the problem, that he also addressed in a paper published in the current proceedings. Several people also discussed or exchanged with us about subtyping and polymorphism: Zhiwu Xu, Véronique Benzaken and Kim Nguyen.

References

- [1] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proceedings of the VLDB Endowment*, 3(1):906–917, 2010.
- [2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th international conference on functional programming (ICFP '03)*, pages 51–63, Uppsala, Sweden, 2003. <http://doi.acm.org/10.1145/944705.944711>.
- [3] G. M. Bierman, A. D. Gordon, C. Hrițcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th international conference on functional programming (ICFP '10)*, pages 105–116, Baltimore, MD, USA, 2010. <http://doi.acm.org/10.1145/1863543.1863560>.
- [4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C recommendation, 2007.
- [5] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th international conference on functional programming (ICFP '11)*, Tokyo, 2011.
- [6] J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- [7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems (TACAS '08)*, pages 337–340, Budapest, 2008. http://dx.doi.org/10.1007/978-3-540-78800-3_24.
- [8] D. Engovatov and J. Robie. XQuery 3.0 requirements, W3C working draft, 2010. <http://www.w3.org/TR/xquery-30-requirements>
- [9] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008.
- [10] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *Proceedings of the 28th conference on programming language design and implementation (PLDI '07)*, pages 342–351, San Diego, CA, USA, 2007. <http://doi.acm.org/10.1145/1250734.1250773>.
- [11] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. <http://doi.acm.org/10.1145/767193.767195>.
- [12] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27:46–90, 2005. <http://doi.acm.org/10.1145/1053468.1053470>.
- [13] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. *ACM Transactions on Programming Languages and Systems*, 32(1):1–56, 2009.
- [14] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
- [15] J. Vouillon. Polymorphic regular tree types and patterns. In *Proceedings of the 33rd symposium on principles of programming languages (POPL '06)*, pages 103–114, Charleston, SC, USA, 2006. <http://doi.acm.org/10.1145/1111037.1111047>.
- [16] P. Wadler. Theorems for free! In *Proceedings of the 4th international conference on functional programming languages and computer architecture (FPCA '89)*, pages 347–359, London, 1989. <http://doi.acm.org/10.1145/99370.99404>.